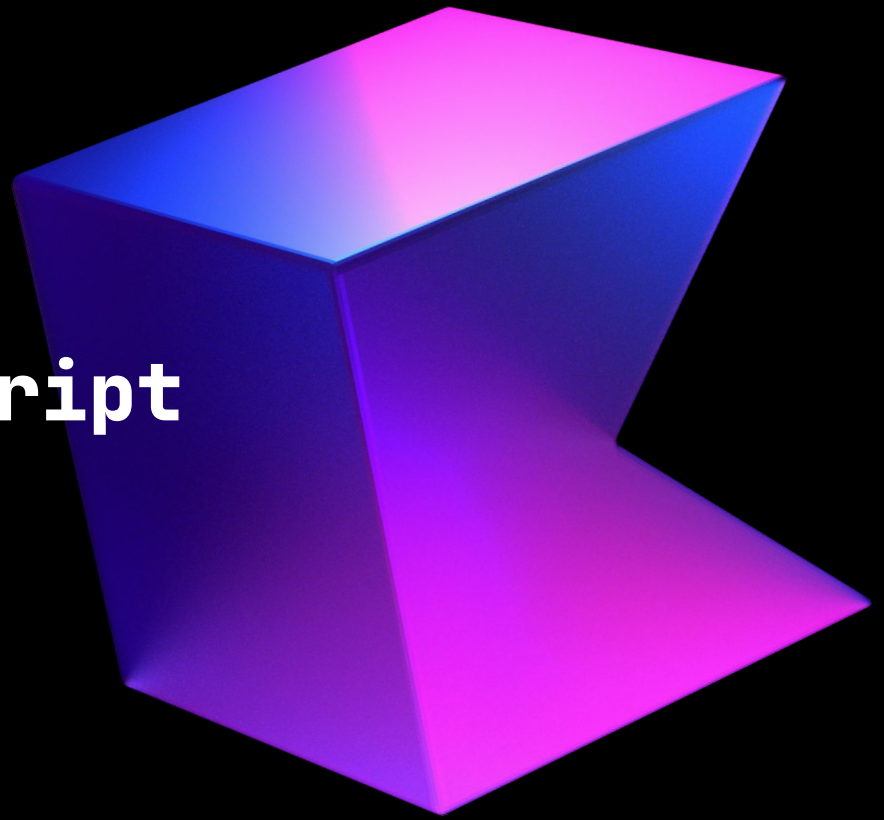
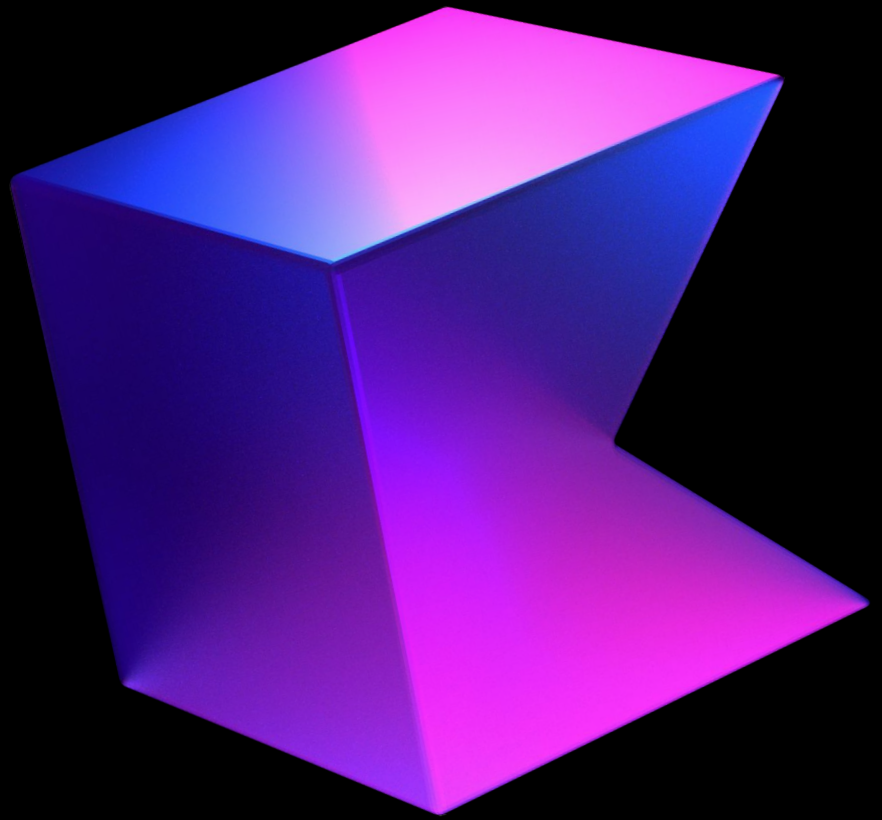
JAVA ≠ Javascript



Kotlin

Introduction to Kotlin



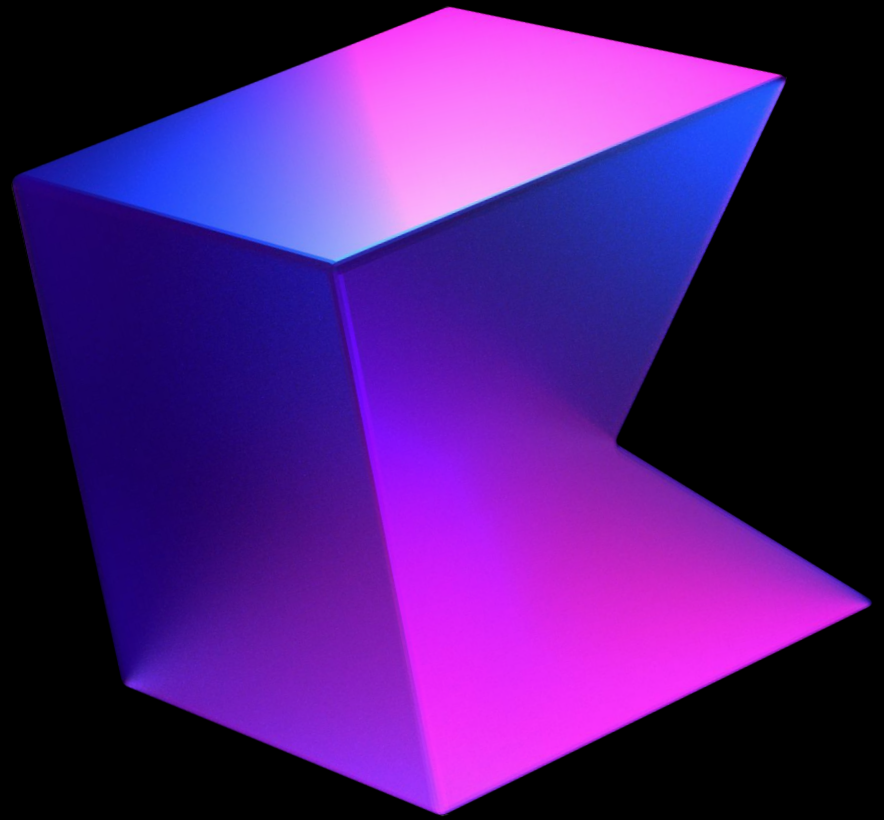
@kotlin

Agenda

Kotlin Overview
- Introduction
  - The Basics
  - Basic Types
  - Collection
  - Variables
  - Mutable vs Immutable
  - String Templates
  - Class vs Data Class
  - Functions
  - If, When, Operation, Loop, Ranges
  - Null Safety
  - Elvis Operator
  - Safe Call
  - Unsafe Call
- Scope Function

Spring Boot Overview

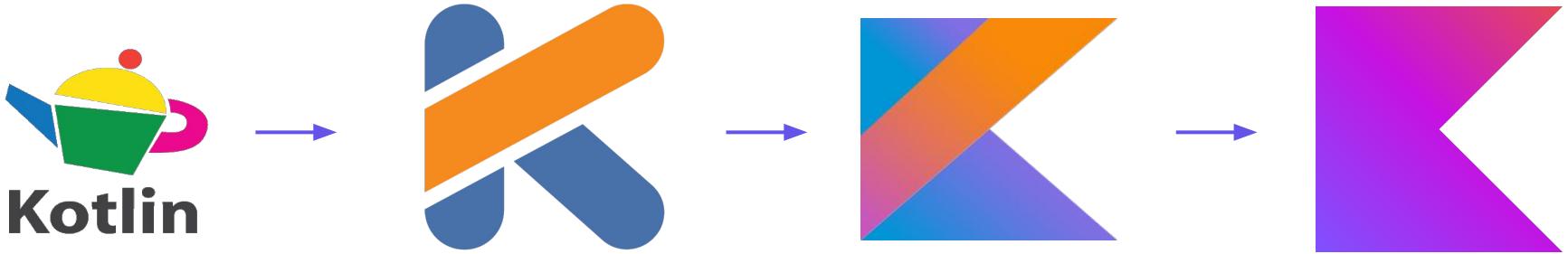
Workshop



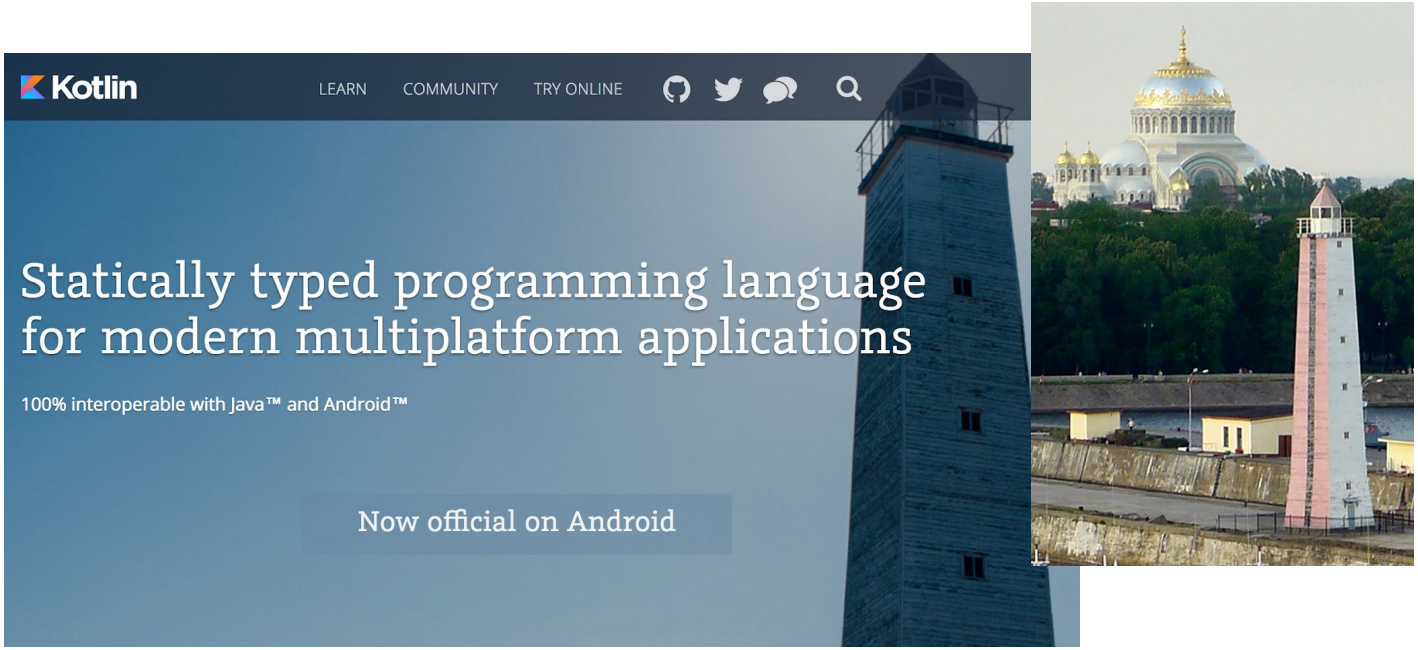
Why Kotlin?

- Expressiveness/Conciseness
- Safety
- Portability/Compatibility
- Convenience
- High Quality IDE Support
- Community
- Android 👀
- More than a gazillion devices run ~~Java~~ Kotlin
- Lactose free
- ~~Sugar free~~
- Gluten free

Logo



Name



Kotlin is named after an island in the Gulf of Finland.

The basics

```
fun main(args: Array<String>) {
    println("Hello, world!")
}
```

```
public static void main(String []args) {
     System.out.println("Hello, world!");
}
```

Hello, world!

```
fun main(args: Array<String>) {
        println("Hello, world!")
}

fun main() {
        println("Hello, world!")
}

fun main() = println("Hello, world!")
```

Where is “;”???

Basic Types

- Int
- Long - Bigger Int
- Float - With decimal (.)
- Double - Bigger Float
- String
- Boolean
- Arrays

Collection

Collection type	Description
Lists	Ordered collections of items
Sets	**Unique** unordered collections of items
Maps	Sets of key-value pairs where keys are unique and map to only one value

```
list.add(5)
list.add(6)
list.add(1)
```

List
5
6
1

```
set.add(1)
set.add(5)
set.add(6)
set.add(1)
```

Set
1
5
6

Key	Value
key_1	value_1
key_2	value_2

```
x.get("key_1") // value_1
```

Variables

`val/var myValue: Type = someValue`

- `var` - mutable
- `val` - immutable
- Type can be inferred in most cases

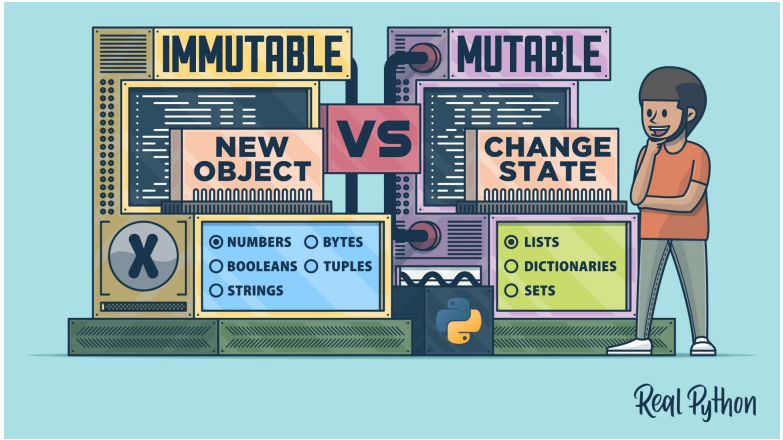
`var = variable`
`val = valued`

```
val a: Int = 1   // immediate assignment

var b = 2        // 'Int' type is inferred
b = a            // Reassigning to 'var' is okay
```

Mutable vs. Immutable

Mutable = able to be change.

Immutable = unchanging over time or unable to be changed.



Variables

```
const val/val myValue: Type = someValue
```

- `const val` - compile-time const value
- `val` - immutable value
- for `const val` use uppercase for naming

```
const val NAME = "Kotlin" // can be calculated at compile-time
val nameLowered = NAME.lowercase() // cannot be calculated at compile-time
```

String templates

```
val i = 10
val s = "Kotlin"

println("i = $i") // “i = 10”
println("Length of $s is ${s.length}")

System.out.println("Length of “ + s + “ is “ + s.length)
```

class vs. data class

`class`

- As usual class, used to contain functionality, data, etc.

`data class`

- Designed to be data holder
- Easily compared between 2 data classes
- Support `toString()` for easily debug/logging
- `copy()` function

```
data class Person(var name: String, var age: Int)

...

val person1 = Person("Toptoppy", 19)

val person2 = Person("Toptoppy", 19)

person1.equals(person2) //true

...

println(person1.toString()) //Person(name="Toptoppy",age=19)

...

val person3 = person1.copy("Methi")

person1.equals(person3) //false

println(person3.toString()) //Person(name="Methi",age=19)
```

Functions

```
fun sum(a: Int, b: Int): Int {
     return a + b
}

private fun mul(a: Int, b: Int) = a * b

fun printMul(a: Int, b: Int): Unit {
     println(mul(a, b))
}

fun printMul1(a: Int = 1, b: Int) {
     println(mul(a, b))
}

fun printMul2(a: Int, b: Int = 1) = println(mul(a, b))
```

Single expression function.

`Unit` means that the function does not return anything meaningful.

It can be omitted.

Arguments can have **default** values.

If expression

```
fun maxOf(a: Int, b: Int): Int {
      if (a > b) {
            return a
      } else {
            return b
      }
}
```

is the same as

Be careful!

```
fun maxOf(a: Int, b: Int) =
      if (a > b) {
            a
      } else {
            b
      }
```

`if` can be an expression (it can return).

Can be a one-liner:

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

If expression

There is no short-if in Kotlin!

When expression

```
when (x) {
    1 → print("x == 1")
    2 → print("x == 2")
    else → {
        print("x is neither 1 nor 2")
    }
}
```

`when` returns, the same way that `if` does.

```
when {
    x < 0 → print("x < 0")
    x > 0 → print("x > 0")
    else → {
        print("x == 0")
    }
}
```

The condition can be inside of the branches.

When statement

```
fun serveTeaTo(customer: Customer) {
      val teaSack = takeRandomTeaSack()

      when (teaSack) {
          is OolongSack -> error("We don't serve Chinese tea like $teaSack!")
          in trialTeaSacks, teaSackBoughtLastNight ->
                 error("Are you insane?! We cannot serve uncertified tea!")
      }

      teaPackage.brew().serveTo(customer)
}
```

`when` can accept several options in one branch. `else` branch can be omitted if `when` block is used as a *statement*.

&& vs and

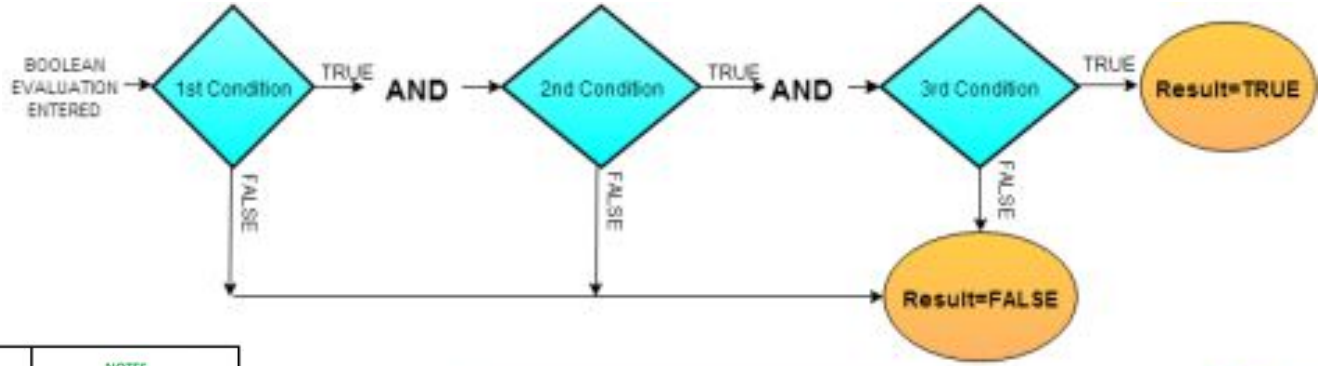
```
if (a && b) { ... }     VS     if (a and b) { ... }
```

Unlike the `&&` operator, this function does not perform **short-circuit evaluation**.

The same behavior with OR:

```
if (a || b) { ... }     VS     if (a or b) { ... }
```

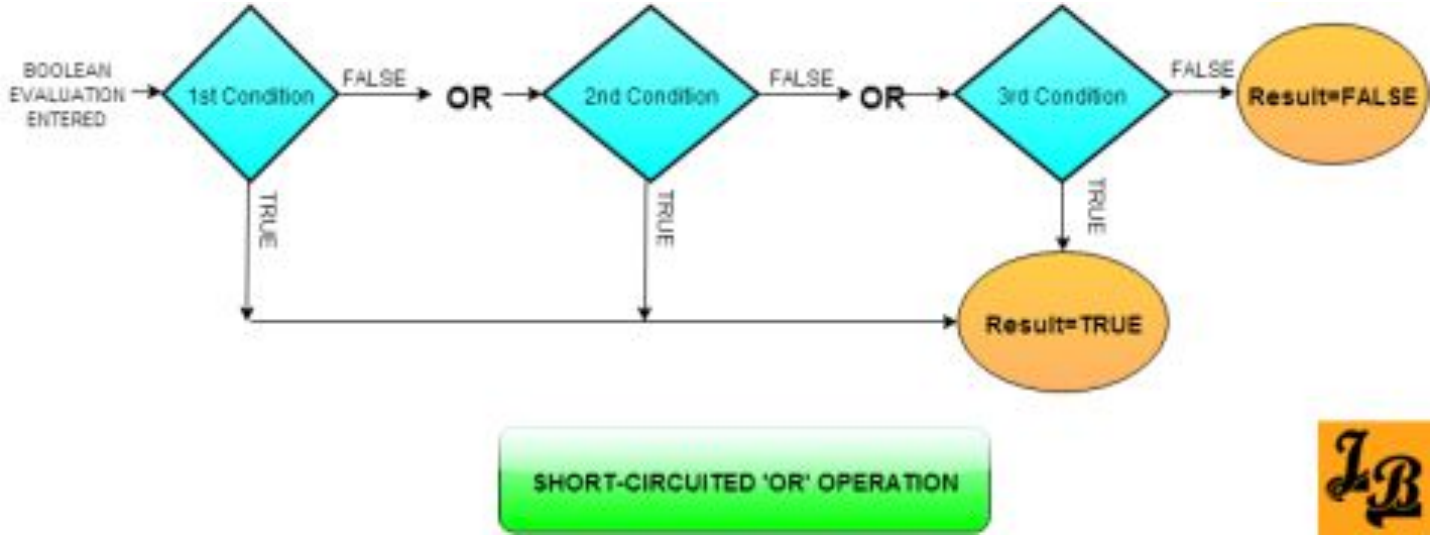
Short-Circuit Evaluation (&&)



OPERATION	RESULT	NOTES
X or Y	If X is False, then Y, else X	Y is executed only if X is False Else if X is true, X is result.
X and Y	If X is false, then X else Y	Y is executed only if X is true, else if X is false , X is result.
not X	if X is true, then false, else true	not has lower priority than non - boolean operators. Eg. not a==b => not (a==b)

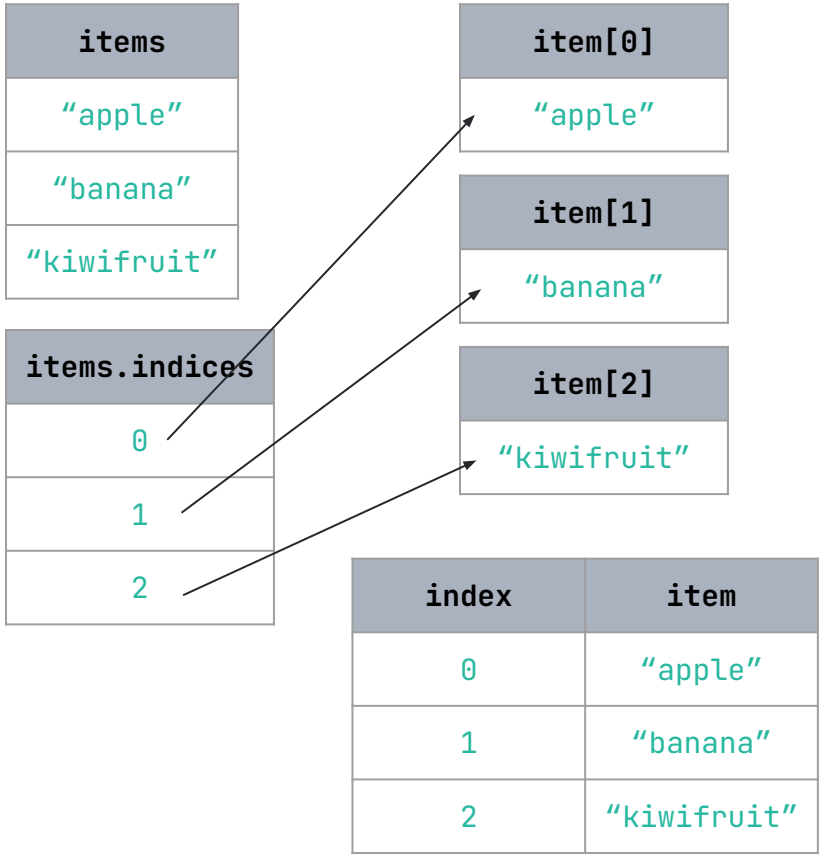
SHORT-CIRCUITED 'AND' OPERATION

Short-Circuit Evaluation (||)



Loops

```
val items = listOf("apple", "banana", "kiwifruit")

for (item in items) {
    println(item)
}

for (index in items.indices) {
    println("item at $index is ${items[index]}")
}

for ((index, item) in items.withIndex()) {
    println("item at $index is $item")
}
```



Loops

```
val items = listOf("apple", "banana", "kiwifruit")

var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}

var toComplete: Boolean
do {
    ...
    toComplete = ...
} while(toComplete)
```

The condition variable can be initialized inside to the `do…while` loop.

Loops

There are `break` and `continue` labels for loops:

```
myLabel@ for (item in items) {
    for (anotherItem in otherItems) {
        if (...) break@myLabel
        else continue@myLabel
    }
}
```

Ranges

```
val x = 10
for (x in 1..5) {
    print(x)
}

if (x in 1..10) {
    println("fits in range")
}

for (x in 9 downTo 0 step 3) {
    print(x)
}
```

`downTo` and `step` are extension functions, not keywords.

`'..'` is actually `T.rangeTo(that: T)`

Null safety

```
val notNullText: String = "Definitely not null"
val nullableText1: String? = "Might be null"
val nullableText2: String? = null

fun funny(text: String?) {
      if (text ≠ null)
            println(text)
      else
            println("Nothing to print :(")
}
```

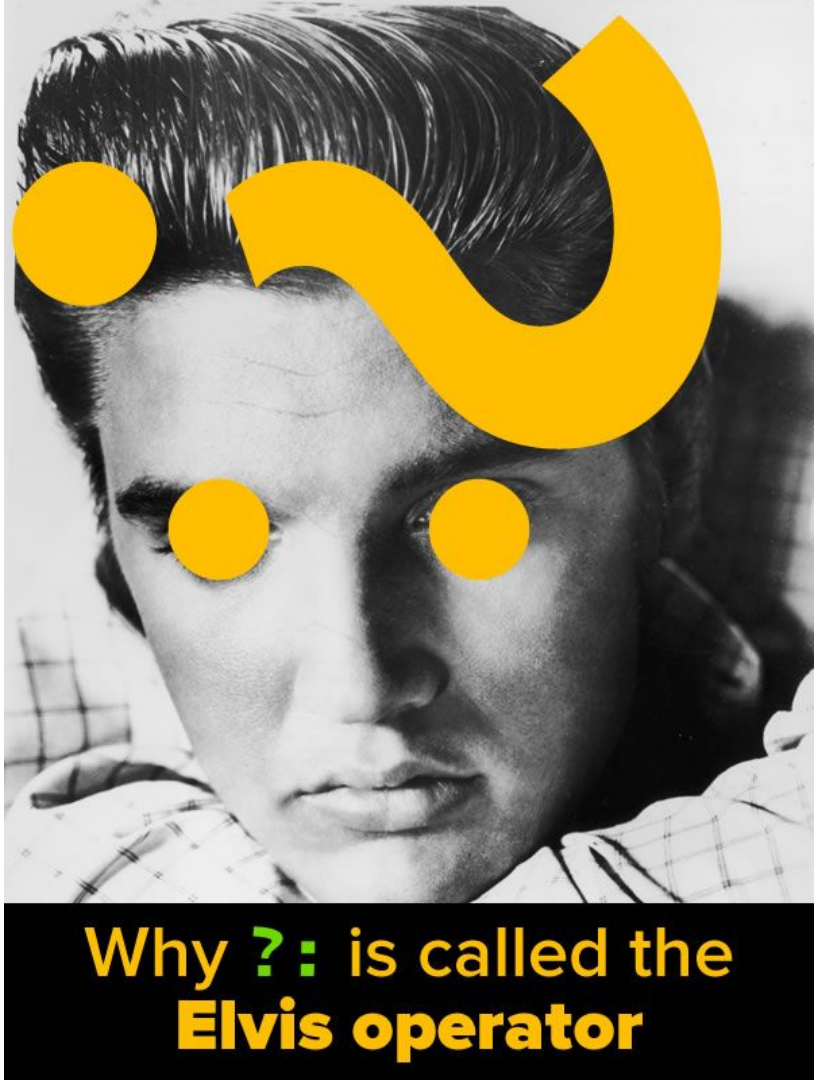
Elvis operator

Elvis operator ?:

```
fun funnier(text: String?) {
      val toPrint = text ?: "Nothing to print :("
      println(toPrint)
}
```


```
 fun funny(text: String?) {
       if (text ≠ null)
             println(text)
       else
             println("Nothing to print :(")
 }
```

Fun Fact ;)



Safe Calls

```
val a: String? = “Kotlin”
val b: String? = null
println(b?.length) → [redacted]
println(a?.length) → [redacted]
```

It won’t return NPE (NullPointerException)

To print only for non-null values, you can use the safe call operator together with `let`:

```
employee.department?.head?.name?.let { println(it) }
```

Unsafe Calls

```
val b: String? = null
println(b!!.length)
```

The not-null assertion operator (`!!`) converts any value to a non-null type and throws an NPE exception if the value is null.

Please, **avoid** using **unsafe** calls!

TODO

Always throws a NotImplementedError at **run-time** if called, stating that operation is not implemented.

```
// Throws an error at run-time if calls this function, but compiles
fun findItemOrNull(id: String): Item? = TODO("Find item $id")


// Does not compile at all
fun findItemOrNull(id: String): Item? = { }
```

Lambda expressions

```
val sum: (Int, Int) → Int = { x: Int, y: Int → x + y }
val mul = { x: Int, y: Int → x * y }
```

According to Kotlin convention, if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses:

```
val badProduct = items.fold(1, { acc, e → acc * e })

val goodProduct = items.fold(1) { acc, e → acc * e }
```

If the lambda is the only argument, the parentheses can be omitted entirely (the documentation calls this feature "trailing lambda as a parameter"):

```
run({ println("Not Cool") })
run { println("Very Cool") }
```

Scope functions

By definition, Scoped functions are functions that execute a block of code within the context of an object.

Why?

*Scope functions make code more **clear**, **readable**, and **concise** which are Kotlin language's main features.*

Types of scope functions

1. ***let :*** working with nullable objects to avoid NullPointerException.
2. **run :** operate on a nullable object, executing lambda expressions.
3. ***with :*** operating on non-null objects.
4. ***apply :*** changing object configuration.
5. ***also :*** adding additional operations.

Summary Scope Function

Here is a short guide for choosing scope functions depending on the intended purpose:

- Executing a lambda on non-nullable objects: `let`
- Introducing an expression as a variable in local scope: `let`
- Additional effects: `also`
- Object configuration: `apply`
- Object configuration and computing the result: `run`
- Running statements where an expression is required: non-extension `run`
- Grouping function calls on an object: `with`

Returns / Reference to receiver	Receiver	Results of lambda
it	also	let
this	apply	run/with

let

Usually working with nullable objects to avoid **NPE**

Expect if fieldA is have value will print `Hello ${value}` but if it is null, ignore.

```
val fieldA: String? = null

if (fieldA ≠ null) {
     println(“Hello ${fieldA}”)
}

fieldA?.let { println(“Hello $it”) }
```

```
class Person() {
    var name: String = "Abcd"
    var contactNumber: String = "1234567890"
    var address: String = "xyz"
}

val p1 = Person().let {
    "The name of the Person is: ${it.name}"
}

print(p1)
```

Question - Which value it will print?

```
employee.department?.head?.name?.let { println(it) }
```

also

Used to perform additional operation on an initialize object , Call “ and also do this following with the object ”

```
val numbers = mutableListOf("one", "two", "three")
numbers
    .also { println("The list elements before adding new one: $it") }
    .add("four")
```

apply

Used for changing an object configuration, the definition of apply is “Apply the following assignment to this object”

```
data class Person(
        var name: String,
        var age: Int = 0,
        var city: String = "")
```

```
val adam = Person("Adam").apply {
    this.age = 32
    this.city = "London"
}
```

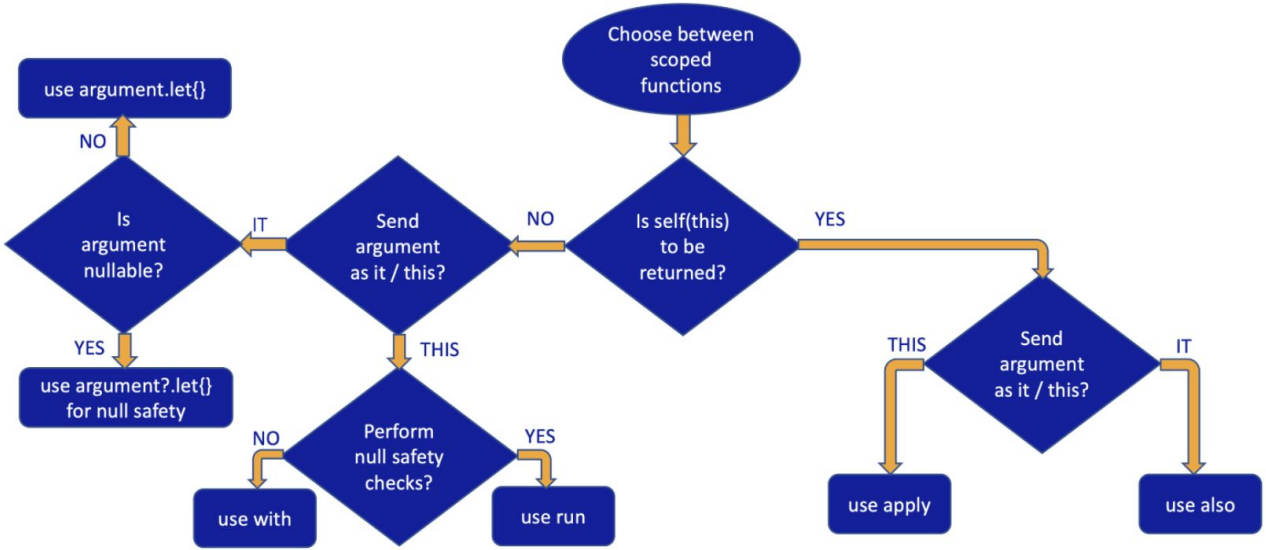
```
val adam = Person("Adam")
adam.age = 32
adam.city = “London”
```

RUN

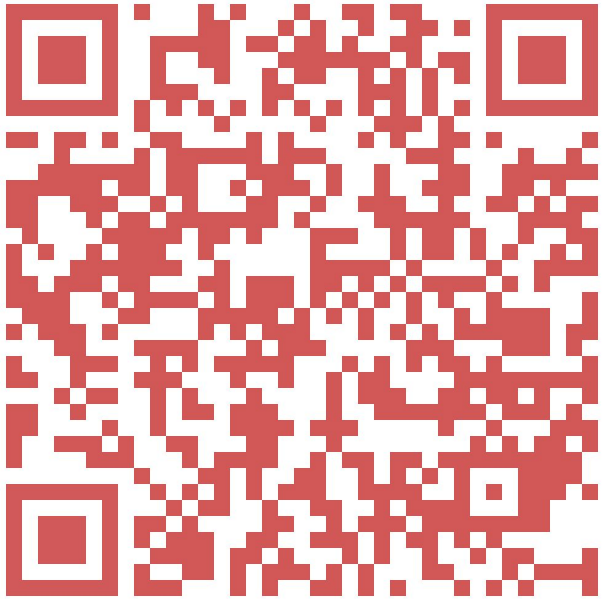
```
val service = MultiportService("https://example.kotlinlang.org", 80)

val result = service.run {
    port = 8080
    query(prepareRequest() + " to port $port")
}

// the same code written with let() function:
val letResult = service.let {
    it.port = 8080
    it.query(it.prepareRequest() + " to port ${it.port}")
}
```

With

```
val numbers = mutableListOf("one", "two", "three")
val firstAndLast = with(numbers) {
   "The first element is ${first()}," +
   " the last element is ${last()}"
}
println(firstAndLast)
// The first element is one, the last element is three
```


Choose between scoped functions
Is self(this) to be returned?
NO
YES
Send argument as it / this?
IT
THIS
Is argument nullable?
NO
YES
use argument.let{}
use argument?.let{} for null safety
Perform null safety checks?
NO
YES
use with
use run
Send argument as it / this?
THIS
IT
use apply
use also

Want more?



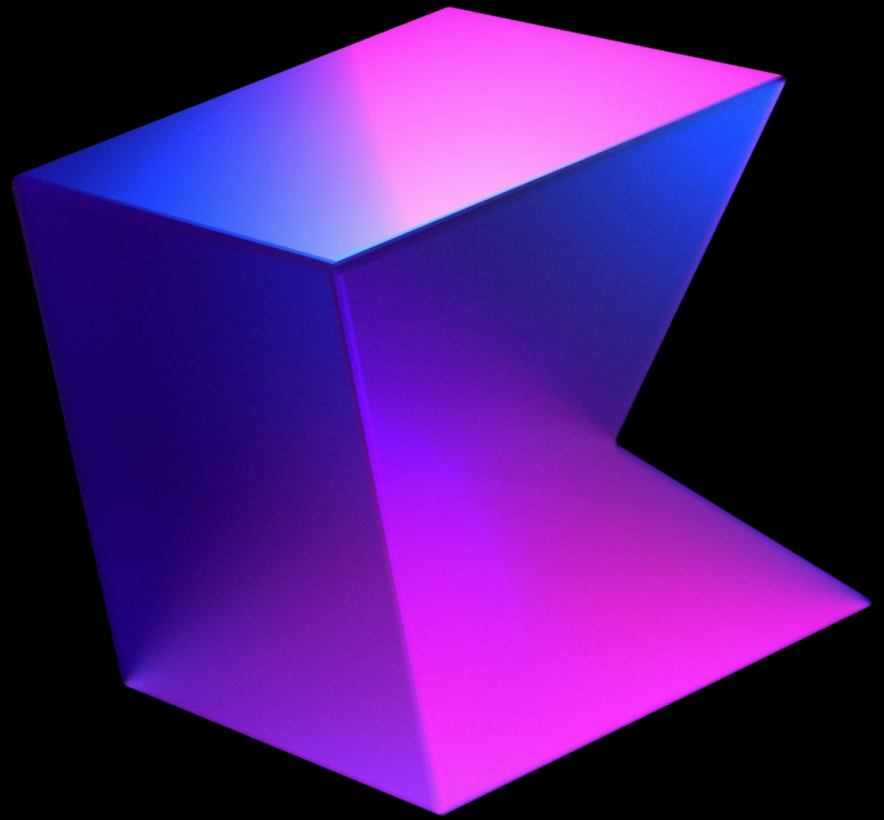
When in doubt

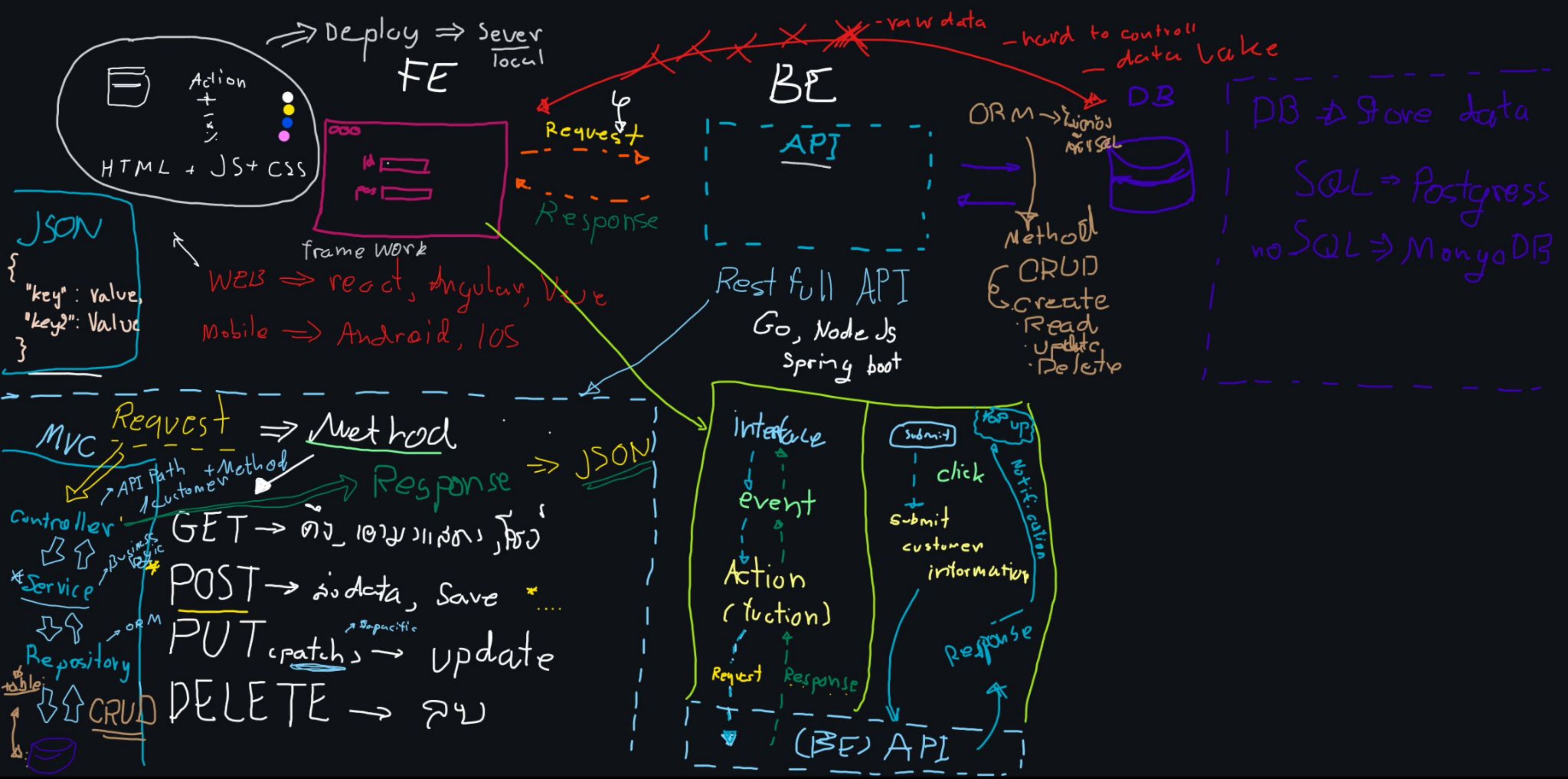
Go to:

- kotlinlang.org
- kotlinlang.org/docs
- play.kotlinlang.org/byExample

GO TO SOFAR ROOM

Q&A

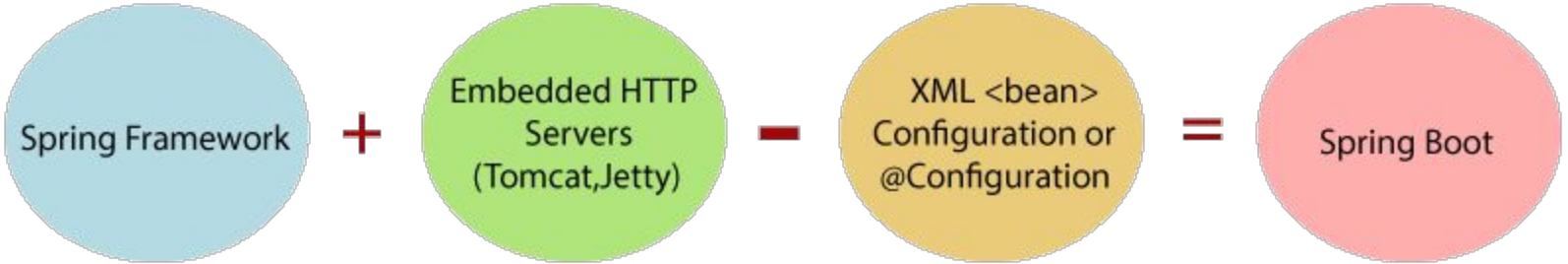




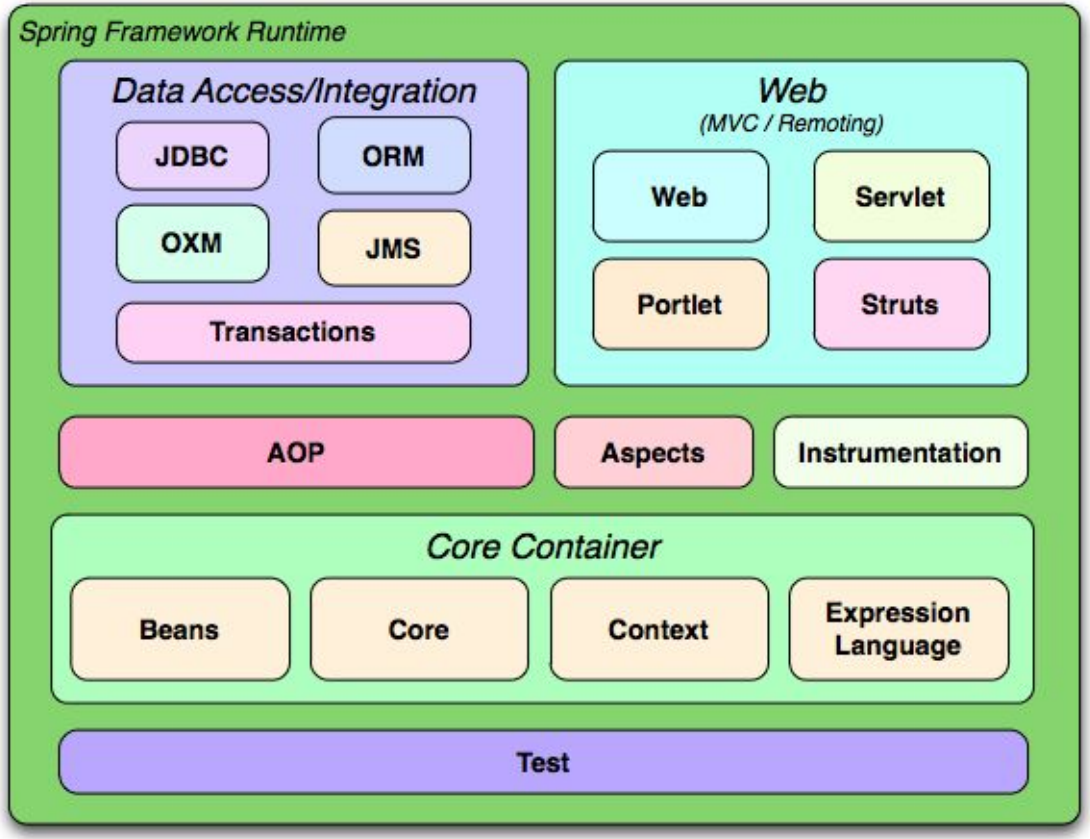
Deploy ⇒ Sever / local
Action
HTML + JS + CSS
FE
frame work
Request
Response
BE
API
-raw data
-hard to controll
- data lake
DB
ORM
Method
CRUD
Create
Read
update
Delete
DB ⇒ Store data
SQL ⇒ Postgress
no SQL ⇒ MongoDB
JSON
{
"key" : value,
"key2": value
}
WEB ⇒ react, Angular, Vue
Mobile ⇒ Android, IOS
Rest full API
Go, Node Js
Spring boot
MVC
Request ⇒ Method
API Path + Method
/customer
Response ⇒ JSON
Controller
Service
Business logic
Repository
ORM
table
CRUD
GET → ดึง เอามาแสดง, โชว์
POST → ส่ง data, Save
PUT (patch) → update
specific
DELETE → ลบ
interface
event
Action
(fuction)
Request
Response
Submit
click
submit customer information
Pop up
Notification
Response
(BE) API


spring
boot

What is Spring Boot



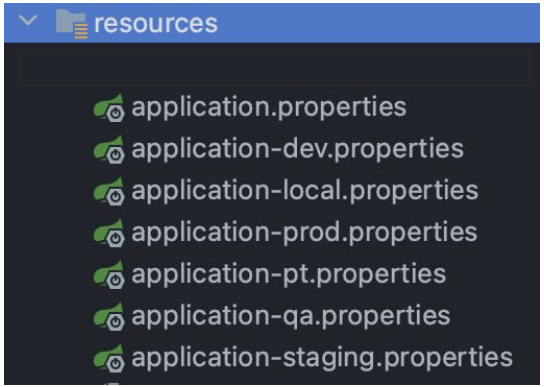
Spring Framework



Key Highlight of Spring Boot

- **Autoconfiguration**
- **Creates stand-alone application that can be run using jar file.**
- **Embedded Servers**
- **Easier to connect with Databases**
- **Dependency Injection**
- **A lot of starter libraries for implementation**
- **Compatible with JAVA libraries of Spring Boot**
- **Spring Profile**
- **And Many More!**

Spring Profile



- Provide ability to separate configuration for each environment.
- Support hierarchy so we can reduce duplicate configuration.
- Support `.properties` and `.yml` file

application-prod.properties
prod db connection

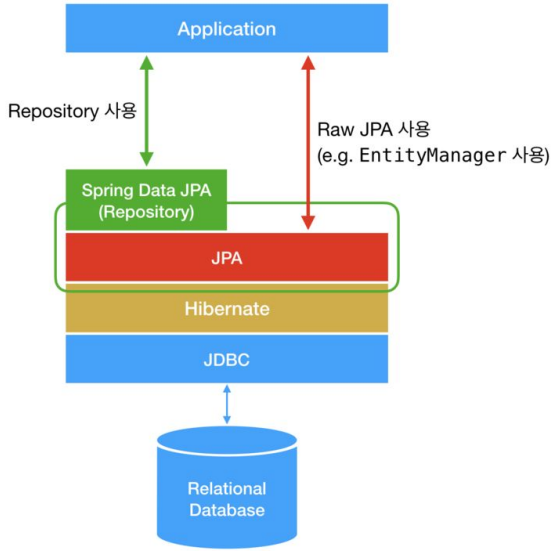
application-dev.properties

application.properties
common db connection

JDBC
Directly manages the database access with SQL queries.
Works with database tables and columns directly.
Requires manual schema creation and update.
Tends to be faster as it involves direct SQL queries.

ORM
Abstracts the database access with objects.
Objects are mapped to database tables.
Can automate schema creation and update.
Tends to be slower due to the additional abstraction layer.

Spring Data JPA



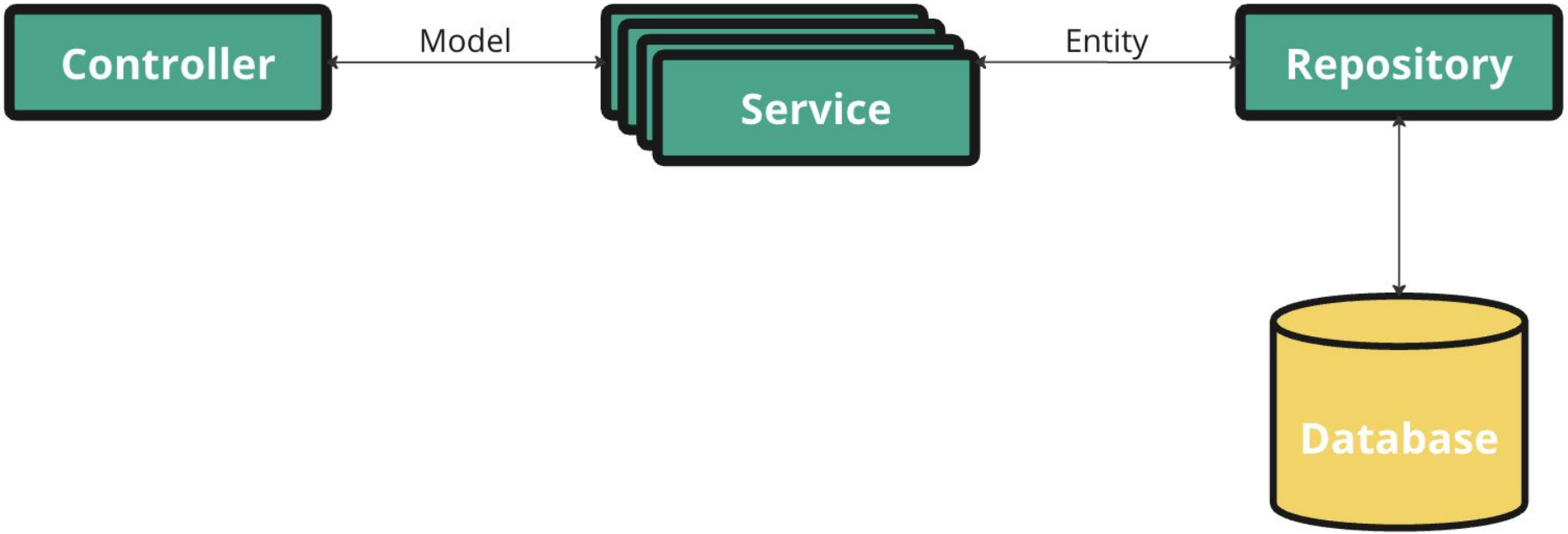
- Provide an easier way to connect to database for spring.
- Low code (or no code at all) to CRUD (Create, Read, Update, Delete)

Example

```
fun findAll()
fun findBy<field_name>(value: <T>)
fun findByCustomerType(customerType: String): CustomerEntity
fun findAllByCustomerType(customerType: String): List<CustomerEntity>
fun findAllByCustomerTypeOrderByRegisterDate(customerCode: String): List<CustomerEntity>
```

```
package com.                .repositories

import com.                .entity.CustomerEntity
import org.springframework.data.jpa.repository.JpaRepository
import org.springframework.stereotype.Repository

@Repository
interface CustomerRepository : JpaRepository<CustomerEntity, String>
```

Project Structure




<html>
<head>
JAVA Script
</>
$i<=$min;
PHP
Workshop


<html>
<head>
JAVA Script
Q & A
</>
C++
<DIV>
$i<=$min;
PHP

Resources

- https://kotlinlang.org/education/
- https://www.baeldung.com/